Flappy Bird using Reinforcement Learning

Yi Zhou
yzhou20@ucla.edu

Rong Jin
rongjin98@ucla.edu

Tianyu Xiang
txiang@ucla.edu

Weitao Sun
weitaosun@ucla.edu

***Abstract*—Reinforcement learning (RL) is a machine learning training method using rewarding and punishments for behaviors. In past decades, reinforcement learning's attention has been drawn increasingly and it has begun to demonstrate its potential applications in the real life. Reinforcement learning is one of the most effective ways to develop artificial intelligence systems autonomously, and is essential for applications where it can provide sequence of decisions. In this project, we showed that reinforcement learning is very effective in learning how to play the game Flappy Bird. Through many adjustments, we finally obtained considerable results using Q-learning method in OpenAI Gym environment, showing that it can have a performance that surpasses ordinary players, and discussed the difficulties and potential improvements of reinforcement learning.**

***Index Terms*—Flappy bird, Reinforcement Learning, Q-learning, OpenAI Gym.**

I. Introduction

Flappy Bird is an intuitive 2-dimensional mobile game developed by Dong Nyugen in 2013 [8]. At the beginning of the game, there will be a bird moving all the way to the right side of the screen, and will sink due to gravity to form a parabolic motion. When people touch the screen, the bird will flap its wings and move up. Players need to keep the bird flying by tapping the screen at the accurate moment and avoid obstacle pipes. As the bird crosses a pipe, player get a one-point boost, and when the bird hits the pipe, the bird dies and the game ends automatically. The goal is to get the highest possible score, and therefore people need to keep the bird out of the pipes. At the same time, the difficulty of the game will gradually increase, because as the game progresses, the speed of the bird's flight will increase.

We build the game into a mathematical model that can be solved according to the rules of the game. We set the bird as the agent and the pipeline as the environment. We will reward the bird when it goes through the pipe, and punish it when it hits the pipe. After trying several different approaches, we decided to use Q-learning. The goal of Q-learning is to find the optimal policy that maximizes the expected sum of future rewards [3]. Playing games is particularly challenging because our goal is to provide only pixel information and score mediators. We applied this project in the OpenAI Gym environment and compared with the previous work, we mainly modified two files to better perform recording and replaying. The first one is the game logic and the second one is that the game environment executing scripts. By testing Q-learning algorithm in 2-dimensional space, the result does not satisfy our expects. Therefore, we extend to 3-dimensional space to train the data and using biased epsilon decay, a expected higher result was achieved.

II. Previous Work

This section touches on two subjects. Flappy bird environments people have previously used, and algorithms that have been tested on these environments.

A learning environment enables engineers to focus solely on the algorithm design. Previously, PyGame-Learning-Environment (PLE) is selected in many literature and lab studies. PLE mimics the Arcade Learning Environment interface, and allows to be programmed in Python. The valid input to the environment is moving up, and when stepped the environment outputs a frame in RGB [5]. However, it is argued as a naive environment because the bird's velocity in PLE is binary. The bird can only go up or down with a definite speed. And that is improved in Openai Gym Environment FlappyBird-v0 (gym) [4]. In gym environment, the bird's environment affects it's velocity, which makes the game non-linear. Besides, additional to the RGB frame output like in PLE, the gym environment also outputs birds relative position to the next pipe. The output is viewed intuitively as an output state.

Markov decision process (MDP) works when the states, actions, system dynamics and rewards are presented. It requires the agent to have a complete knowledge of the system. States and actions are assumed to be complete; however, there is no prove that shows the full knowledge of the system dynamics. The MDP has been tried on PLE and the agent scores an average 65 during 100 episodes [2]. Since then, model free reinforce learning algorithm was used to tackle this problem. Q-learning was able to achieve over 10 million score on PLE [7]. However, the Q-learning in previous attempt does not work in the gym environment because the epsilon function is designed to be decaying with respect to the number of training episodes. And it turns out that in the gym environment the agent is still exploring new states when epsilon was decayed to zero.

III. Formulation

Flappy-Bird is a two dimensional game that is visually displayed on a screen. In order to build an agent that plays the game, the game environment has to be framed into a mathematical model that can be solved with a method. In this section, we talk about how the model is built and Q-learning algorithm.

A. Modeling

Formulating the flappy-bird problem into a viable model helps to design the agent. The core mechanics of flappy-bird is that the bird flaps and tries to survive as long as possible by passing through pipes. It can be interpreted as an agent interacting with the environment, where the bird is the agent and the pipes are the environment.

Environment can be analyzed using concept of the state space. Naively, we assume the state space to be three dimensional. First dimension is horizontal distance of the bird to the center of the pipe. It tells the bird how many more actions can it make before meeting the next pipe. The second dimension is the vertical distance of the bird to the lower pipe, which is the parameter tells the bird to avoid collision. The first two dimensions are directly extracted from the observation of the environment. Since the observation is essentially a float, which leads to infinite statespace, we quantized the observation space by keeping three decimal points. The number of kept decimal points is regarded as the resolution of quantization.

The third dimension is the vertical velocity of the bird. The vertical velocity is highly correlated to the current and previous taken actions and the attack angle of the bird. For example, a large positive vertical velocity implies consecutive flap actions, vice versa. The reason for including vertical velocity as another dimension of the statespace is to let the agent be aware of configurations of the bird when passes through or hits pipes. Initial velocity is always assumed to be zero. State space is formulated in equation 1, where x, y, v represent the three dimensional state space.

$$S = \{(x, y, v) \big| x, y, v \in \mathbb{Z}\} \tag{1}$$

The agent's action choices are limited. At any given moment, the bird can only choose to flap or do nothing. Either action will step the environment. Hence, the action space in flappy-bird is discrete. The flap or no flap action can be described as 0 or 1 as shown in equation 2.

$$A = \{0, 1\} \tag{2}$$

Reward function aims to teach agent how to behave. The goal is to make the bird survive as long as possible. More specifically, we reward the bird every time it passes a pipe. Although passing the pipe is not given from the environment, we can induce by observation the state transitions. Besides, penalize when it hit the pipe as shown in 3.

$$R(s) = \begin{cases} 10 & s.t. \;\; ||s_t + 1||_2 >> ||s_t||_2 \\ -1000 & s.t. \;\; isDone = True \\ 0 & otherwise \end{cases} \tag{3}$$

However, modeling the system dynamics is hard because pipes, as obstacles, are randomly generated. The environment does give pipe location. So at any time, the agent only knows the pipe center, but does not know the exact position the previous pipe or next pipe. With that, model free learning model is used as shown in figure 1.

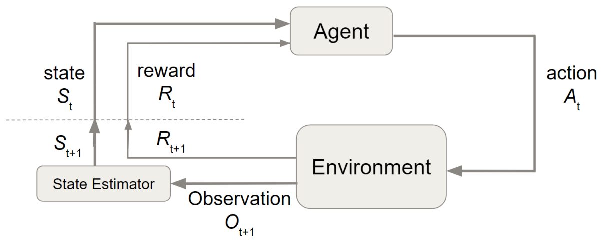

Fig. 1: System model

B. Q-Learning

Reinforcement learning (RL) studies how action taken by intelligent agents affects the environment. If the agent's action directly affects the environment, it is said to be an online problem. If the learning does not based on the modeling, it is said to be a model free learning [6]. From the analysis from the modeling section, we have shown that the flappy-bird is an online model free RL problem. Q-learning is used to tackle this problem. In equation 5, S and A represents the state and action. Q is the table we created to store learned S, A value. α is the learning rate, which is set to be decay over number of episodes that been iterated. γ is the discount factor.

$$Q(s,a) = Q(s,a)+\alpha[R(s,a)+\gamma \max Q'(s',a')-Q(s,a)] \tag{4}$$

The goal of Q-learning is to find an optimal policy that maximize the expected sum of future rewards. A determinate policy π takes in an S and outputs an A. However, the agent does not know the optimal policy from the beginning. To learn such policy, the agent needs to explore and exploit. ϵ-greedy is chosen to be our approach. The idea is to maximize reward by exploitation the best action given the current state while leave room for exploration.

$$\pi(s_{t+1}) = \begin{cases} \arg\min_a Q(s,a) & w.p. \;\; 1-\epsilon \\ a \sim A & w.p. \quad \epsilon \end{cases} \tag{5}$$

ϵ is used to control agent's exploration and exploitation choices. As the agents keeps learning , the agent should be more confident with the next choice. Without decaying , the agent for certain probability will choose a random choice even though it was confident about the policy. So decay function is designed to let the agent choose an optimal policy more frequently. In many literature, decay by number of episodes the agent has played. The logic was that agent should be more confident as it play more games [1]. However, it in a game with large space, the learning rate has already decayed to insignificance before all states are explored. Hence, we design it to be decayed through number of times of a state which the agent has visited. So, the agent can always learn when traversing through a new state as shown in 6. β is a constant that represents decaying rate, and C_s counts how many times a state has been visited.

$$\epsilon = \epsilon * \beta^{C_S} \tag{6}$$

The pseudo code is presented in 1.

Algorithm 1 Training flappy bird agent with Q learning

```
Require: $Q(s,a), C(s), \alpha_0, \epsilon_0, \gamma, N$
Ensure:
0: for $i$ in range($N$) do
0:     Reset $S$
0:     $isDone \leftarrow False$
0:     while not $isDone$ (Game is not done) do
0:         $A_t \leftarrow \pi(s_t)$  4
0:         $s_{t+1}, R_t, isDone \leftarrow$ take $A$ on environment
0:         $Q \leftarrow$ update Q function  5
0:         $\epsilon \leftarrow$ update $\epsilon$  6
0:         $C(s_t) \leftarrow C(s_t) + 1$
0:         $S_t \leftarrow S_{t+1}$
0:     end while
   return  $Q$
```

IV. Implementation and Results

A. Environment

In this project, we apply our reinforcement learning algorithm in OpenAI Gym environment of Flappy Bird. OpenAI Gym is an interface provides various game environments for testing and simulating, and formulates these games into MDPs. Specifically, each game environment contains an action space, an observation space and a reward space, which are essentially the key elements of any reinforcement learning problem.

In the Flappy Bird environment, we modified two files in order to perform two crucial functionalities, recording and replaying. The first file is the game logic, which are basically the mechanism of the game. Once the game starts, it generates upper and lower pipes set randomly, updates the states, and checks the collisions. The second one is the game environment executing scripts, which initializes and closes the game, executes the action, gets the observation and renders the frame. In order to replay the simulation we want, we add two additional game logics, one for recording and one for replaying, as well as two additional executing scripts corresponding to respective game logics. The recording game logic and executing scripts can record the complete upper and lower pipes generation sequence and the whole time history of taken actions. Since the transition of states by a taken action is deterministic in Flappy Bird, we can replay any specific episode of simulation after getting these time histories of information. Then the replaying game logic loads the recorded pipe generation sequence, and generates the pipe set in order instead of random generation, while the replaying executing script executes the action from the saved time history of taken actions.

For this project, we used the original game logic and executing pair for training. After getting the trained Q matrix, we loaded Q matrix to the recording pair. Inside the recording pair, the game is simulated by the trained agent for multiple times until we get a satisfying result, such as achieving 3000 points.Then, we exported saved action and pipe generation sequence of that simulation to the replaying pair where actions and pipe generations are picked strictly following the order of sequences. Since the rendering consumes a lot of unnecessary time which heavily hinders the training process, we only render the satisfactory simulation during the replaying process. The overall pipeline is shown below:

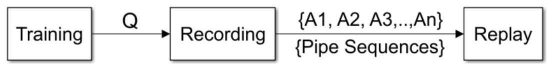

Fig. 2: Training, Simulating and Replaying Pipeline

B. Results

In this project, we tested our q-learning algorithm for one 2-dimensional statespace and one 3-dimensional statespace, as well as two different strategies of epsilon decay. We recorded the accumulative summation of scores for every 10 thousand episodes as the performance return to display the learning tendency.

1) 2D Statespace: The 2D-statespace only consists of (x,y) pair in 1. ϵ decays at a power of 0.99 for each episode. We trained the agent for two hundred thousand episodes.

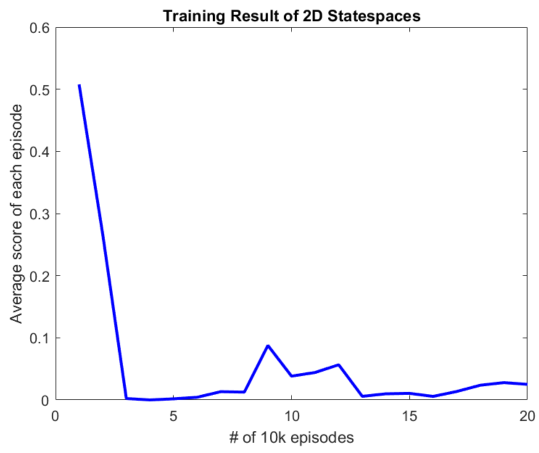

Fig. 3: Return tendency of 2D statespace

From the figure, only the first 10 thousand episodes display some learning progress which is about an average of 0.5 score per episode. Then the algorithm quickly overestimates the values which cause the a rapid decrease of performance return all the way to zero. The rest of simulation becomes trivial since the policy becomes deterministic as the number of episodes increases while the performance of trained agent has no difference than completely random actions. By the end of training, the total state-space size is 58368.

2) 3D Statespace: Since the training result of 2-dimensional statespace is not quite satisfying, we extended one more dimension to the statespace, which is shown in 1. Same as the one in 2D cases, ϵ decays at a power of 0.99 for each episode.

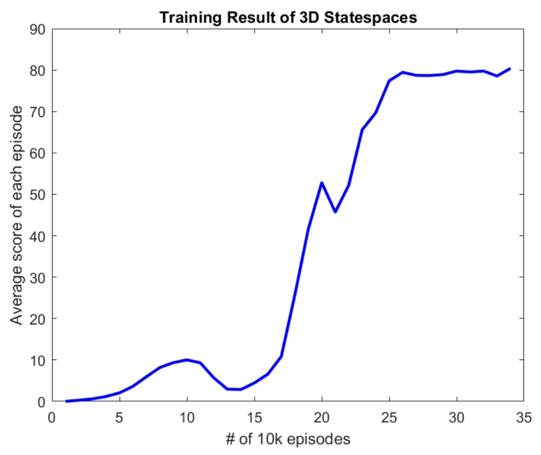

Fig. 4: Return tendency of 3D statespace

From the figure, the training performance of 3-dimensional state space is way much better than the 2-dimensional case. The agent eventually reaches at average score of 80 points after 350 thousand episodes. However, as we can notice the performance return curve becomes flat after 250 thousand episodes, and the final statespace size is 220451 which is only about 4 times larger than the 2-dimensional case. Since the size of velocity state space must be larger than 4, it means that many states are not explored by the agent. In fact, current strategy of epsilon decay makes the agent avoid exploring very quickly as the episode increases. For example, the 10 thousand power of 0.99 is about $2.25 * 10^{-44}$ which essentially equals to zero. Therefore, theoretically, the agent stops exploration as soon as after the first 10 thousand episodes. Yet, the first 10 thousand episodes do not contain much effective exploration since in most of cases the agent either collides with the pipe or ground very quickly. Therefore, more exploration are definitely required for better performance.

3) Biased Epsilon Decay: In order to encourage exploration of the agent, we make change the strategy of unbiased epsilon decay to biased epsilon decay. Specifically, instead of making all the states share a common epsilon which decays every episode, each state has its own common epsilon and decays for each time that state is visited. In this case, the agent will exploit at the states that are often visited, and explore more at rare states. Meanwhile, rare states that are previously exploited, may be actually more lucrative than the usual states.

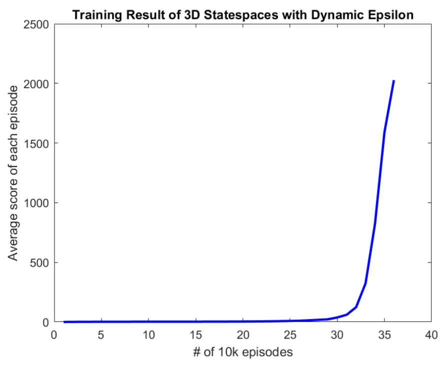

Fig. 5: Return of 3D statespace, biased epsilon decay

The figure above displays a very slow learning tendency at the beginning which matches with our expectation, since the agent needs to explore through more states before learning the optimal policy at that state. After 300 thousand episodes, the performance return increases almost exponentially and reaches at an average of 2000 points for each episode after 360 thousand episodes of training. By the end of training, the final statespace size increases to 587312 which is about 2.5 times larger than the previous one.

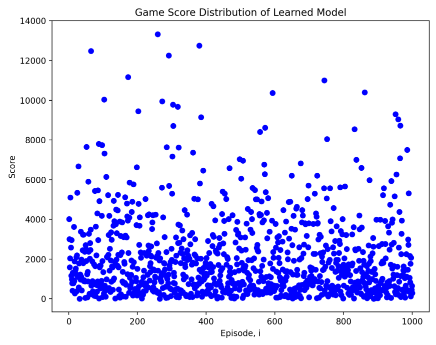

Fig. 6: Score distribution by trained agent of 1000 episodes

Then we let the trained agent to play the game for one thousand times and get the score distribution above. We can notice, many scores stack at the region of 1000 and 2000 points. The maximum score achieved during the simulation is around 14000 points.

The training result of using 3D statespace with biased epsilon decay is satisfying. However, we do notice the agent now almost always collides the pipe at a configuration, which it hits at the edge of old upper pipe, when it actually observes next pipes. In this configuration, our agent failed to detect the pipe just but not fully passed, which basically makes it kind of a blind-spot of our agent with our current setting of states.

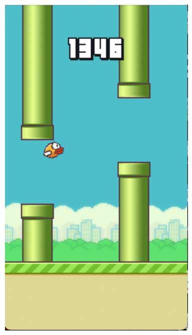

Fig. 7: Blind spot of the agent

V. Conclusion and Future Works

A. Summary

The score we achieved on Flappy Bird using Reinforcement Learning is much great than a human player. And by using biased epsilon decay, convergence speed slows down but it greatly extends the exploration region, and improves final learning performance. However, there are still many uncertainties regarding to the blindspot and the effectiveness of our algorithm.

B. Future works

As we mentioned, the agent currently almost always stuck at a certain type of configuration, which we temporarily regarded as a "blindspot". However, we could not verify if this configuration is really a blindspot or it is just merely an extremely rare condition which could be solved by more episodes, since at 350k episodes of training, each 10k more episodes needs more than 8 hours to complete without rendering(as the average score is 2000 at this point). Therefore, the uncertain identity of "blindspot" leads to three studying directions for future works.

1) More Episodes for Tendency Observation: We can simply train the agents for more episodes so that we can observe a more complete performance tendency over episodes. Currently, our training process stopped at 360,000 episodes, and the performance return still displays a sheer increase tendency according to Fig.5. After sufficient amount of episodes, we should be able to observe the flattening of performance return curve, similar as the one in Fig.4. And if it is a blindspot, we should observe that the performance return after certain amount of episodes gradually decreases because that blindspot should keep giving incorrect feedback to the agent due to the incomplete information of the state space. However, this process will be quite time consuming. One million training episodes could take hundreds or thousands of hours to complete.

2) Add One More Dimension to Statespace: We can add another dimension that measures the vertical distance to the upper pipe. By adding this dimension, ideally, the agent should pass pipes in the middle of the gap. However, due to the curse of dimensionality, an additional dimensional state may lead to more than 100 million possible states, since the current statespace size is already half of a million. Such large statespace will inevitably lead to an extremely slow learning process. We probably could not observe new learning progress within one million episodes of training. To address this problem, we could probably reduce the resolution from 3 decimal points to 2 decimal points which essentially shrinks the size of statespace by a factor of 100. But clearly, this approach is also very time-consuming and might lead to nowhere.

3) Use DQN for Function Approximation: Deep Q-learning algorithm is widely used for solving video games learning problems, and have proven its effectiveness. In our project, we did not actually use function approximation, but used quantization to reduce the infinite statepspace to finite statespace. On the other hand, DQN accepts the frames of the game as an input and use a series of convolutional neutal networks and fully-connected neural networks to approximate the Q-value function. To implement DQN, input image might need some preprocessing to reduce many unnecessary fectures of the game image. In the future, we could implement an DQN and compare its performance with our current Q-learning algorithm.

References

[1] Robert Chuchro and Deepak Gupta. "Game playing with deep q-learning using openai gym". In: *Semantic Scholar* (2017).

[2] *MDP Q-Learning SARSA Shan-Hung Wu DataLab*. https://nthu-datalab.github.io/ml/labs/16_Q-Learning/16_Q-Learning.html. Accessed: 2021-12-10.

[3] Zachary Hervieux-Moore Moritz Ebeling-Rump Manfred Kao. "Applying Q-Learning to Flappy Bird". In: *Semantic Scholar* (2016).

[4] Gabriel Nogueira. *flappy-bird-gym*. https://github.com/Talendar/flappy-bird-gym. 2021.

[5] P.W.D. Norman. *PyGame-Learning-Environment*. https://github.com/ntasfi/PyGame-Learning-Environment. 2019.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[7] *Towards Data Science Reinforcement Learning in Python with Flappy Bird*. https://towardsdatascience.com/reinforcement-learning-in-python-with-flappy-bird-37eb01a4e786. Accessed: 2021-11-15.

[8] Rhiannon Williams. "What is flappy bird? the game taking the app store by storm". In: *telegraph* (2014).